(2)

AD-A237 789

NTATION PAGE

Form Approved
OMB No. 0704-0188

	RT DATE	3. REPORT TYPE AND DATES COVERED FINAL 15 Dec 88 to 14 Jun 90

4. TITLE AND SUBTITLE
DATA COMPILATION: ITS DESIGN ABD ANALYSIS

5. FUNDING NUMBERS
AFOSR-89-0186
61102F 2304/A2

6. AUTHOR(S)
JOHN FRANCO, DANIEL P. FRIEDMAN

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
INDIANA UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
BLOOMINGTON, INDIANA 47405

8. PERFORMING ORGANIZATION REPORT NUMBER
AFOSR-TR- 91 0548

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR/NM
Bldg 410
Bolling AFB DC 20332-6448

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFOSR-89-0186

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release;
distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The aim of this research is to study the idea compilation and its impact on the development of concise, efficient, verifable code. This entails developing, formalizing, analyzing, and extending a data compilation methodology based on work proposed. One goal of our study is to delineate the scope of applicability of data compliation techniques. Our purpose is similar to that of researchers studying functional transformations and partial computation.
Regarding data compilation, the greatest success of this work has been the appriction of the ideas; a varitey of problems using extend-syntax in Scheme. The solutions we have obtained are concise, are of optimal complexity, and yet are relatively free of data structure considerations including boundedness and sparsity. The potential for solving a wider variety of problems in this style by adding features to Scheme has been shown to be great. We have proposed some new features and modifications to existing features which will be needed to manage data compilation more efficently.

14. SUBJECT TERMS			15. NUMBER OF PAGES
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

UNITED STATES AIR FORCE
AIR FORCE OFFICE OF SCIENTIFIC RESEARCH
BUILDING 410, BOLLING AFB, D.C. 20332

Grant No. AFOSR 89-0186

FINAL SCIENTIFIC REPORT

December 1988 to June 1990

Data Compilation: It's Design and Analysis

John Franco, Daniel P. Friedman, Principal Investigators

Department of Computer Science
Indiana University
Bloomington, Indiana 47405

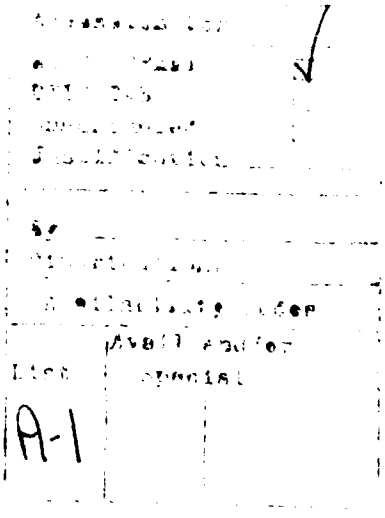

TABLE OF CONTENTS

1. Introduction and Research Objective

Traditionally, software developers have seen data as an entity to be entirely processed by a program. However, it is possible to view data as instructions for building at least some of the program text. For example, the string matching algorithms of [2,16] syntactically transform input strings to control structures that become part of the computational process. As is the case in string matching, programs in which data is compiled can be efficient because they use instance-specific information to tailor themselves to "best fit" the problem instance offered. However, the design of such programs may be complicated by the additional information needed to specify how to optimally compile input data. We know that with the right language and tools it is possible to develop concise programs for data compilation which have theoretically optimal complexity for certain classes of problems. How far can these tools be extended to provide a means for generating concise and optimally efficient data-compiled programs for a multitude of problem classes?

The aim of this research is to study the idea of data compilation and its impact on the development of concise, efficient, verifiable code. This entails developing, formalizing, analyzing, and extending a data compilation methodology based on work proposed in [8]. One goal of our study is to delineate the scope of applicability of data compilation techniques. Our purpose is similar to that of researchers studying functional transformations and partial computation.

Functional programming has been developed with the aim of writing easily verifiable programs (we call this comprehensibility). Comprehensibility is achieved, for the most part, by attenuating considerations of state and control. The problem with functional techniques is that programs so written, in conventional style and for conventional hardware, usually are neither as efficient as possible nor understood in terms of their time and/or space complexity. One possible remedy is the concept of transforming a functional program to an equivalent, possibly non-functional program of improved complexity [3,4,5,13,22]. The transformation itself can be interactive. The idea is to have a programmer produce a comprehensible program and let the transformation take care of the complexity issues. The transformation need not be terribly efficient since it is performed only once, just prior to compilation.

Although remarkable successes have been achieved with transformations, there are several stumbling blocks to this approach as it is now practiced. First, one or more "eureka" steps is required in order to achieve success. Second, although the problem of producing syntactic transformation rules such as "append → rplacd" seems tractable, the problem of producing transformation rules which are semantically generated is a hard one to solve correctly since an enormous amount of rule interaction must be taken into account. For

example, consider the problem of producing an efficient Union-Find algorithm. It is easy to find an optimal algorithm for doing Union and an optimal algorithm for doing Find, but neither of the two is optimal for solving the Union-Find problem [1]. Third, the transformation approach cannot be complete since the problem of determining complexity is undecidable [20].

Partial computation of a computer program is by definition "specializing a general program based upon its operating environment into a more efficient program." The objective is to reduce run-time complexity by re-using the results of previous partial computations. The idea is to regard programs as data and perform an analysis which allows computation to proceed as much as possible with no or partial actual data. Analyzed programs may be manipulated in order to continue the partial computation as much as possible.

Our approach is different from both the partial computation and functional transformation approaches. Partial computations regard programs as data to be manipulated. Data compilation, however, *regards data as programs.* Doing so results in some form of partial computation in many cases. Functional transformations convert functional programs to efficient, possibly non-functional programs. The transformations require one or more "eureka" steps. Data compilation also requires a "eureka" step of sorts to develop a program that replicates in precisely the right way. But, it relies on the creative effort of the programmer to design a solution of low complexity. We believe our tools make this creative effort relatively easy because the metaphors behind their use will change little from one problem to the next.

One problem with the data-compilation approach is that the time to compile a program plus data must be taken into account when measuring run-time complexity. Ordinarily, compile-time is regarded as unimportant since only one compile is necessary using conventional programming techniques. But, compile-time may be significant under any data-compilation methodology if re-compilation is necessary for each change in data (as is the case for many graph theory problems). Compile-time may not be significant if there is both a compiled-data argument and a non-compiled data argument. For example, consider the problem of simulating a finite state acceptor. Such simulations require a description of the machine and one or more inputs to the machine as data. The description data is compiled but no other data is. Hence, re-compiles are only necessary when machine descriptions change and compile-time can be amortized over all simulations of the same machine. This property is similar to that which motivates research on partial computation (see e.g., [7,12,14,21,23]). In summary, compile-time must be combined with run-time for a true impression of complexity. Part of this research aims to quantify this combination more precisely so that the complexity of data-compiled programs can be compared with the complexity of more conventional programs.

2. Overview of Results

In [8,9] we present a programming style that is the basis for a proposed programming methodology which syntactically transforms some or all input data to compiled code. The transformation may occur as late as run-time. The methodology is particularly suited to problems that can be solved by subproblem decomposition. Each subproblem is regarded as a three-state object which communicates with other objects. Generally, but not exclusively, the communications are requests for the solution or partial solution to more primitive subproblems. The states are 1) waiting for the first request 2) processing that request, and 3) having found the solution to the subproblem. The objects are modeled as procedures and each state change results in a change in procedure definition. The programmer specifies only a template for procedure definitions and communications links. That template is used to transform (at least some of) the data to an interacting network of procedures, possibly changing with time as invocations of subproblems *implied* by the data occur.

Since objects change state, corresponding network procedures are self-modifying. However, the modifications are restricted by the fact that a template is required to specify them, that there are only two state changes, and that the code of each state can appear to be functional. Furthermore, the modifications are only a computational gimmick and the semantics of the network procedures *do not change* between states. Therefore, it seems that the strongly imperative nature of the programs developed with the methodology may actually have a manageable semantics.

The methodology is dependent on the Scheme programming language (see [6] for a description of Scheme). This is partly because some implementations of Scheme already contain useful tools and features which can support some aspects of a data-compiled methodology. For example, the macro-expansion facility known as `extend-syntax` (see [18,19,17]) can be useful in constructing communications links which occur at compile-time although it currently cannot deal with links that must be constructed at run-time. As another example, the control abstraction known as `call-with-current-continuation` or `call/cc` for short, originally designed to manage collateral evaluation of a multitude of processes, can be used to construct communication links at run-time. As a third example, tail-recursion is handled in Scheme without stacking. This feature is important because procedures defined in our methodology often do not return. Scheme handles such cases without wasting space. Finally, we mention that although many conceptual "copies" of the same procedure may be produced by our programs, these all occupy the same physical space in a computer except for local state. Examples of the methodology in Scheme are given in the next section.

Scheme lacks full run-time capability for creating communication links. Thus, certain

problems do not have an elegant data-compiled solution in Scheme. In [9] we propose a facility for dynamic mutual recursion which, if adopted, would allow efficient and concise code for a large class of problems including many from operations research which are normally solved by dynamic programming techniques. An example is given in section 4.

However, the problem of communicating streams, even those constructed at run-time, may be solved using `call/cc`. In [10] we show how to construct stream networks in Scheme with `call/cc`. Demand-driven code based on this facility can have improved theoretical efficiency, and be pseudo-functional (semantics of the streams are invariant in the presence of assignment) as is shown in [10]. We elaborate on this in section 5.

Application of the methodology to complicated real problems was anticipated. A natural application is in logic programming where implementations such as Prolog need to take advantage of partial computation if they are to be competitive. Our intention was to build a Warren Abstract Machine with the tools available and attempt extensions as a next step. This ambitious task was not completed within the funding period.

During the reporting period, we wrote a chapter for a book editied by John Feo of Lawrence Livermore Labs which is a comparison of parallel programming languages [11]. This chapter includes Scheme solutions to four benchmark problems. While this work is not directly related to the mission of the project, it should be important to the community.

3. Examples of Data Compilation in Scheme

In this section we provide three simple examples of data-compilation using communication links constructed at compile-time. The objective of this section is only to provide a taste of the methodology. Other examples may be found in our publications.

3.1. A Finite State Acceptor

The first example is a program for simulating the finite state acceptor mentioned in the first section. Although this program does not completely illustrate our ideas, it is a good one to start with. A brief description follows the code for the benefit of the reader who is not familiar with Scheme.

```
(extend-syntax (fsa)
  [(fsa init-state ([Sa final [a Sb] ...] ...))
   (letrec
     ([Sa (lambda (l)
            (case (car l) [a (Sb (cdr l))] ... [$ final] [else #f]))]
      ...)
     init-state)])
```

In `fsa` the procedures `Sa`, `Sb`, and `final` represent states, `l` is a list of input symbols, `a` is a symbol from the input alphabet, and `$` is a special string termination symbol. We show how this program works by means of the following instantiation:

```
(define test-fsa
  (lambda ()
    (let ([machine
           (fsa
             q0
             ([q0 #t [a q0] [b q1]]
              [q1 #t [b q1] [a q2]]
              [q2 #f [a q2] [b q2]]))])
      (machine '(a a a b b a $)))))
```

This input is based on the following finite state machine (Q is the set of states, Σ is the alphabet, δ is the transition function, F is the set of final states, and $q0$ is the start state):

$$Q = \{q0, q1, q2\}$$
$$\Sigma = \{a, b\}$$
$$\delta(q0, a) = q0,\ \delta(q0, b) = \delta(q1, b) = q1,\ \delta(q1, a) = \delta(q2, a) = \delta(q2, b) = q2$$
$$F = \{q0, q1\}.$$

Some of the data given in `test-fsa` describes the finite state machine. This data is compiled by `fsa` (defined in the first section of code) into a finite state acceptor and given the name `machine`. In `test-fsa` the line `[q0 #t [a q0] [b q1]]` is a pattern that matches the expression `[Sa final [a Sb] ... ]` in the definition of `fsa`. The symbol `#t` (matching `final`) means state `q0` (matching `Sa`) is a final state, the pair `[a q0]` (matching `[a Sb]`) means on input symbol `a` jump to state `q0`, and the pair `[b q1]` (matching `...`) means on input symbol `b` jump to state `q1`. The next two lines have the same pattern and match, in similar fashion, the second ellipsis (`...`) in the second line of the definition of `fsa`. The first `q0` in `test-fsa` matches `init-state` and represents the starting state of `machine`.

The machine itself is actually a collection of procedures, one per state, which are defined by the letrec in the definition of fsa. The procedures are named after their corresponding states and take as input a string of input-symbols, the first one being the next-input-symbol. Each procedure is a single case statement with one case for each pairing of input-symbol and state. With a match on the next-input-symbol, the procedure corresponding to next-state is invoked with what remains of the input string after stripping the next-input-symbol from it. Invocations continue until the symbol $ is the next-input-symbol. Then a #t is returned if and only if the current state of machine is a final state. All procedure invocations are tail-recursive so Scheme does not waste space by stacking them.

The list of input symbols in test-fsa are data which is not compiled. In test-fsa the string '(a a a b b a $ is a list if input symbols. With this list the procedure invocations are as follows: q0, q0, q0, q0, q1, q1, q0. Additional invocations of machine on other lists of input symbols are possible.

The following shows how machine is constructed from the definition of fsa for the particular input given in test-fsa.

```
(letrec
  ([q0 (lambda (l)
         (case (car l) [a (q0 (cdr l))] [b (q1 (cdr l))] [$ #t] [else #f]))]
   [q1 (lambda (l)
         (case (car l) [b (q1 (cdr l))] [a (q2 (cdr l))] [$ #t] [else #f]))]
   [q2 (lambda (l)
         (case (car l) [a (q2 (cdr l))] [b (q2 (cdr l))] [$ #f] [else #f]))])
  q0)
```

From this we see that three procedures representing states are defined by the letrec and the procedure representing the start state (q0) is the value given to machine.

In this example a simulator is built once for a specified machine and the behavior of the machine on each of any number of input strings may be observed. The partial computation leading to the simulator can therefore be amortized over arbitrarily many inputs making the compile-time small compared to total run-time. If none of the data is compiled, run-time is increased since machine specification would have to be redone for each input string. Thus, data compilation results in a theoretical improvement in performance for the problem of simulating a finite state acceptor.

3.2. Topological Sort

Perhaps the most interesting example of our methodology is the following solution to the problem of topologically sorting a partial order.

```
(extend-syntax (topo)
  [(topo ([n (m ...)] ...))
   (letrec ([n (lambda ()
                 (set! n (err-fun 'n))
                 (m) ...
                 (output 'n out-list)
                 (set! n (lambda () #t)))]
            ...)
     (n) ... )])
```

This may be used to sort the following partial order:

```
(topo ([a (b c e)] [b (d f)] [c (b)] [d ()] [e (b f)] [f (d)]))
```

where `[a (b c e)]` means "output" the elements `b`, `c`, and `e` before the element `a`. The list `(b c e)` is called the adjacency list for `a`.

This example shows the three-state nature of the methodology. There is one procedure for each element in the partial order. When a procedure `n` is invoked for the first time (that is, an element is visited), it is redefined by the first `set!` to return an error if it is re-invoked before outputting its name (that is, before putting itself in the total order by executing the line `(output 'n out-list)`); this will happen if there is a cycle and therefore no partial order. But, even while the procedure is redefined, execution due to the first invocation continues and the procedures in the adjacency list of the element represented by `n` (in the line `(m) ...`) are invoked with the result that each of these outputs its name before `n` does. Finally, `n` is redefined again to a procedure that returns `#t`. Thus, further invocations of `n` *do not* generate a recomputation of the results already obtained by the first invocation. The last line (that is the line `(n) ...`) drives the sort by insuring that every procedure is invoked at least once.

This solution has several interesting characteristics. First, it has no tests. Thus, it performs at run-time about as fast as any program for topological sorting possibly can. Second, it achieves maximum efficiency without maintaining data structures which are known to the user. The difference between this solution and the solution presented by Knuth [15] is striking in this regard. Third, with a slight modification in the code, mainly replacing `letrec` with `set!`, it is possible to make incremental changes in the communications structure. Thus, there is no need to recompile all of the data whenever a change is needed. Fourth, the meaning of each procedure does not change throughout the computation. What does change is the procedure's response to an invocation depending

on when that invocation occurs. Fifth, the elements defined in the instance above are not quoted because they are used as *actual* procedures. That is, the data are the procedures.

One question left open in this solution is what does it mean to "output" the name of an element? Clearly, the way we answer this question may have a big impact on how or even whether we can re-write the code without destroying the properties of maximal efficiency, conciseness, and verifiability. In section 5 we show how to construct a stream network so that any stream-consumer procedure can invoke a very slightly modified `topo` in such a way that the element names produced from a given partial order are output by `topo` on demand. This multi-channel stream facility not only supports I/O in the methodology but also brings the benefits of stream processing to it.

3.3. Doctors' Office Simulation

Our last example is a simulation of a doctors' office. Input is a list of doctors, a list of patients, a receptionist, and an event-manager. The event-manager is needed because Scheme does not have the native ability to keep track of concurrent events and their order. Patients stay well for awhile, then get sick, go to the doctor's office, get paired with a doctor when one becomes available, stay with the doctor for some time (no one else can see a doctor that is paired with a patient), are declared well, released, and the cycle repeats. The sickness and wellness times are decided by a procedure called `rand` except in the case of the `stop` patient which is given a time equal to the simulation period. When the `stop` patient becomes sick the simulation terminates. The receptionist does all the bookkeeping needed for pairing doctors with patients.

The code below creates a procedure for each person involved in the simulation. Since there are four different kinds of people, there are four classes of procedures defined. For simplicity, statements involving the accumulation of statistics are left out. Procedures for managing the doctor-list, managing the patient-queue, managing the event-queue, and for testing whether `patient-&-doctor-waiting?` are omitted also. A sample invocation with six patients and three doctors is

```
(doc ((jim joe vena pete jill ann) (lori alice sam) (donna jerry))).
```

This example illustrates how local state is managed by individual procedures. The receptionist keeps a patient-queue and doctor-list which no other procedure knows about. The event-manager keeps an event-queue hidden from others. As in the case of the finite state acceptor, most procedure invocations never result in a return. Unlike the previous examples, invoked procedures may take one of several actions depending on values of the parameter `msg-type`. Procedure definitions do not change at run-time. With slight modification, again centered around using `set!` for `letrec`, the code can support the case

wher- doctors and patients may leave or enter the simulation.

```
(extend-syntax (doc)
   ((doc ((patient ...) (doctor ...) (receptionist event-mgr)))
    (letrec
       ([patient
          (let ([time-to-sick 'infinity])
             (lambda (msg-type throw-away)
                (cond [(eq? msg-type 'name) 'patient]
                      [(eq? msg-type 'event)
                       (receptionist 'enqueue-patient patient '*)]
                      [(eq? msg-type 'new-sick-patient)
                       (set! time-to-sick (rand))
                       (event-mgr 'record patient time-to-sick)])))]
        ...
        [doctor
          (let ([time-to-well 'infinity] [treated '()])
             (lambda (msg-type pat)
                (cond [(eq? msg-type 'name) 'doctor]
                      [(eq? msg-type 'assign-patient)
                       (set! time-to-well (rand))
                       (set! treated pat)
                       (event-mgr 'record doctor time-to-well)]
                      [(eq? msg-type 'event)
                       (treated 'new-sick-patient '*)
                       (receptionist 'release-patient treated doctor)])))]
        ...
        [receptionist
          (let ([patient-Q '()] [doctor-list (list doctor ...)])
             (letrec
                ([treat-waiting-patients-if-docs-available
                    (lambda ()
                       (if (patient-&-doctor-waiting? patient-Q doctor-list)
                           (let ([new-patient (car patient-Q)]
                                 [doct (car doctor-list)])
                             (set! patient-Q (cdr patient-Q))
                             (set! doctor-list (cdr doctor-list))
                             (doct 'assign-patient new-patient)
                             (treat-waiting-patients-if-docs-available))))])
                (lambda (action pat doc)
                   (cond [(eq? action 'enqueue-patient)
                          (if (null? doctor-list)
                              (set! patient-Q (append patient-Q (list pat)))
                              (let ([doct (car doctor-list)])
                                 (set! doctor-list (cdr doctor-list))
                                 (doct 'assign-patient pat)))]
                         [(eq? action 'release-patient)
                          (set! doctor-list (append doctor-list (list doc)))
                          (treat-waiting-patients-if-docs-available)])
                   (event-mgr 'next-event '() 0))))]
        [event-mgr
          (let ([event-Q '()])
             (lambda (action person time)
                (cond [(eq? action 'record)
                       (set! event-Q (insert-in-Q event-Q person time)) time]
                      [(eq? action 'next-event)
                       (let ([next-event (caar event-Q)]
                             [next-time (cdar event-Q)])
                         (set! event-Q (update-Q (cdr event-Q) next-time))
                         (if (not (eq? next-event 'stop))
                             (next-event 'event '())
                             (acknowledge 'receptionist 'event-mgr)))])))])

      (event-mgr 'record 'stop 100)
      (patient 'new-sick-patient '*)
      ...
      (event-mgr 'next-event '() 0))))
```

4. Lexically Scoped, Dynamic, Mutual Recursion

In [9] we propose a facility and a language for maintaining it which may be the basis of unbounded, dynamic mutual recursion in Scheme. This facility, called DMRS for Dynamic Mutually Recursive Structures, has several applications within our data compilation methodology. These are: automatic support for unbounded, undeclared, and sparse vectors and arrays; support for a global, dynamic `letrec`; and memoization. A DMRS can be sparse and therefore wastes little space. It removes some of the burdens of writing procedural specifications that are not relevant to functional specifications such as vector boundaries. Some performance is sacrificed for this feature. DMRS is completely random-access so computational data elements that are logically ordered and ordinarily in successive memory locations (arrays) may not have any exploitable physical relationship in DMRS. Thus, quick address calculations relative to a base address are not possible.

The primary purpose for DMRS is to support dynamic mutual recursion in Scheme. This allows our data compilation techniques to be applicable to a greater number of problems. One example, the Partition problem, is given now.

The Partition problem is defined as follows: given a set $A = \{a_1, a_2, ..., a_n\}$ of objects, a weighting function $w : A \rightarrow N^+$, and an integer K, does there exist a subset $A' \subseteq A$ such that

$$\sum_{a \in A'} w(a) = K?$$

This problem may be solved by subproblem decomposition. For example, suppose $K = 6$ and the list $W = \{1,1,3,3\}$ represents the weights of the objects of a given A, we may decompose this into the two subproblems $W_1 = \{1,3,3\}$, $K = 6$, and $W_2 = \{1,3,3\}$, $K = 5$ corresponding to $a_1 \notin A'$ and $a_1 \in A'$, respectively. The answer to the original problem is "yes" if and only if the answer to either of the subproblems is "yes". In this example, the answer to the first subproblem is "yes" and the answer to the second subproblem is "no" so the answer to the original problem is "yes". Solutions to the subproblems may be found by further decomposition until primitive subproblems are reached.

A recursive solution to the Partition problem based on this decomposition and involving DMRS is

```
(define partition
  (lambda (K W)
    (letrec
      ([part
         (make-DMRS
           (lambda (K^ W^)
             (let ([y (DMRS-ref part K^)])
               (cond [(not (eq? y 'undefined)) y]
                     [(eq? K^ 0) #T]
                     [(< K^ 0) #F]
                     [(null? W^) #F]
                     [else (let ([tmp (or (part (- K^ (car W^)) (cdr W^))
                                          (part K^ (cdr W^)))])
                             (DMRS-set! part K^ tmp))])))
           (lambda (K S) (equal? K S))
           (lambda (K) K)
           'undefined)])
      (part K W))))
```

where the list `W` *is the list of object weights.* The form of this solution is similar to the solutions to certain graph problems in [8] using extend-syntax. The extend-syntax macro expansion facility cannot be used to solve the Partition problem efficiently, however, since the subproblems, which are analogous to vertices in the graph problems, are ***not*** known at compile-time and are uncovered only during run-time. The proposed facility makes these subproblems known to all subproblems when they are created. In other words, creation of the DMRS **part** has the same effect as defining its elements using a form of *dynamic letrec*.

For a description of the language of DMRS and other applications the reader is referred to [9].

5. Multi-way Stream Networks

A stream is a partially computed, possibly infinite list. The tail of a stream holds, in suspension, the ***rest of the computation*** for producing the entire list. When a process demands a list token that is not yet known, the computation of the stream-tail is resumed until the token is computed. Then it is suspended and the generated token is passed to the process that requested it. Streams can help reduce computational effort since list tokens

are not produced if not needed. They can also be used to manage multiple infinite lists since it is not necessary to generate an entire list just to see what the first few tokens are.

Because procedure bodies are not evaluated until they are invoked, it is easy in Scheme to build a functional facility to manage streams. However, we have shown in [10] that stream management can be improved substantially with non-functional or imperative techniques. We call streams created in this fashion *imperative streams*. The semantics of imperative streams is the same as their functional counterpart because a stream never deviates from its original specification .

Imperative streams differ from functional streams in two ways. First, imperative streams are built on a facility of Scheme that is known as `call-with-current-continuation`. Doing so allows several streams to share the same token-producing procedure. Thus, multiway stream networks are possible using imperative streams. Second, new tokens may be appended (destructively) to a stream, not only when they are demanded, but also when they are computed in the process of determining the next token demanded for another stream. This property makes imperative streams theoretically more efficient in heap-based systems than typical solutions for many problems including stream splitting.

Another motivation for developing imperative streams is that they simplify communication between processes in our methodology. As an example, consider again the solution to topological sort offered in section 3. With imperative streams this is now

```
(extend-syntax (topo)
  [(topo topo-channel ([n (m ...)] ...))
   (lambda ()
     (open! topo-channel)
     (letrec ([n (lambda ()
                   (set! n (err-fun 'n topo-channel))
                   (m) ...
                   (put-to-channel! 'n topo-channel)
                   (set! n (lambda () #t)))]
                   ...)
       (n) ... (close! topo-channel)))])

(define err-fun
  (lambda (x channel)
    (lambda () (writeln 'cycle-found-at x) (close! channel))))
```

The stream produced by this code may be connected to the following simple procedure which generates all stream tokens and stops. The segment `(topo stream ([a (b c)]`

...)) is a sample instantiation which defines the stream.

```
(define topo-test
  (lambda ()
    (let*
      ([channel (create-channel)]
       [s (connect! channel (topo channel ([a (b c)] [b (d)] [c (b)] [d ()])))])
      (letrec ([print-stream-token
                  (lambda (s)
                     (if (not (null-s? s))
                         (begin
                            (display (car-s s))
                            (print-stream-token (cdr-s s)))))])
        (print-stream-token s)))))
```

In `topo-test` a channel is created by `(let* ([channel (create-channel)] ...))` and it is connected to `topo` by `[s (connect! channel (topo ...))]` where `s` is the stream generated by `topo`. The procedure `print-stream-token` is a simple loop which consumes all the tokens of `s` one by one. The test `(null-s? s)` terminates the loop when the stream is closed by the `close!` expression in `topo`. The token printed on each iteration is the first token of the remainder of the stream. This is accessed by `(car-s s)`. In the next iteration this token is discarded by `(cdr-s s)`.

It is possible for several consumer procedures to access the same channel [10]. These consumers all see the same stream and any newly generated token for one consumer is a newly generated token for all the rest even if some or all of the others have not yet demanded it. Typically, there is a list of generated tokens in a shared channel: at the beginning of the list is a token not yet demanded by one consumer and at the end there is the earliest token not yet demanded by any of the consumers sharing the channel. The advantage of channel sharing over some functional solutions is that each token need not be generated more than once. As an example consider splitting a stream of integers into two streams, one containing only odd numbers and the other only even. The solution below uses multi-way streams. The output of `even-odd` is a cons cell with the even stream in the `car` and the odd stream in the `cdr`.

```
(define even-odd
  (lambda (input-stream)
    (let*
      ([next-token
         (lambda (e-ch o-ch)
           (lambda ()
             (open! e-ch)
             (letrec
                ([switch
                   (lambda (token)
                      (if (even? token) e-ch o-ch))]
                 [split-stream
                   (lambda (s)
                      (cond [(null-s? s) (close! e-ch) (close! o-ch)]
                            [else (let ([token (car-s s)])
                                    (put-to-channel! token (switch token))
                                    (split-stream (cdr-s s)))]))])
                (split-stream input-stream))))]
       [e-chnl (create-channel)]
       [o-chnl (create-channel)]
       [e-stream (connect! e-chnl (next-token e-chnl o-chnl))]
       [o-stream (share-connect! o-chnl e-chnl)])
      (cons e-stream o-stream))))
```

One may use this procedure in the following simple way where the input stream is the stream of Fibonnaci numbers

```
(set!  fib-channel (create-channel))
(set!  fib-stream (connect!  fib-channel (fibonacci fib-channel)))
(define filter-odd-even (even-odd fib-stream))

(set!  even-stream (car-s filter-odd-even))
(set!  odd-stream (cdr-s filter-odd-even))

(car-s even-stream)
2
(car-s (cdr-s even-stream))
8
(car-s (cdr-s (cdr-s even-stream)))
34
```

```
odd-stream
(1 1 3 5 13 21 (() .  #<procedure>) <no-value-yet> .  #<procedure>)
```

This shows that the odd stream is partially built without demand for any odd integers. The procedure `fibonacci` is included below for completeness.

```
(define fibonacci
  (lambda (channel)
    (lambda ()
      (let ([fib (open! channel)])
        (letrec ([f (lambda (s1 s2)
                      (put-to-channel! (+ (car-s s1) (car-s s2)) channel)
                      (f (cdr-s s1) (cdr-s s2)))])
          (put-to-channel! 1 channel)
          (put-to-channel! 1 channel)
          (f fib (cdr-s fib)))))))
```

The reader is referred to [10] for more examples, the code for multi-way streams, and an explanation of that code.

6. Book Chapter: The Scheme Programming Language

During the grant period the principal investigators wrote a chapter on Scheme for a book edited by John Feo of Lawrence Livermore Laboratory. The book is a comparison of parallel programming languages based on coded solutions to four simple but diverse problems. One of them is the simulation of a doctors' office mentioned in section 3. The others are Hamming's problem, the problem of listing all possible products of numbers taken with replacement from a given list of prime numbers; solving a linear system where the A matrix is a skyline matrix; and the problem of finding all isomers of paraffin (chemical composition C_nH_{2n+2}). Olivier Danvy, currently at Kansas State University, assisted in producing a rigorous and informative description of Scheme.

Two of the four problems, the doctors' office simulation and Hamming's problem, have characteristics suitable for data compilation. One such solution to the doctors' office simulation is given in section 3. The chapter offers a simple functional solution to Hamming's problem in Scheme. It uses the property of delaying procedure body evaluation until invocation to construct a facility for managing streams. The solution could be as simple using multi-way streams if support for recursively building multi-way streams were provided. We have not yet looked at providing such support but it is possible that this

capability will be developed in the future.

7. Probabilistic Analysis of Satisfiability Algorithms

We briefly mention some work on Satisfiability algorithms that was completed under the grant. In the summer of 1989 John Franco organized a workshop on Boolean Functions, Propositional Logic and AI Systems at the FAW in Ulm, Germany. Co-organizers were Prof. Dr. Dr. F. J. Radermacher, Ulm, Prof. Dr. M. M. Richter, Kaiserslautern, and Prof. Peter Hammer, New Brunswick, New Jersey. Proceedings are expected to appear shortly.

In addition, two papers were written (see references 2 and 3 in section 9). One is to appear in *SIAM Journal on Computing* and the other in *Discrete Applied Mathematics.* Partial work on other papers was also carried out under the grant. The full list of works is in section 9.

Finally, John Franco has been co-editing a special issue of *Discrete Applied Mathematics* devoted to connections between logic and combinatorics. Some work on this issue was also supported by the grant.

8. Conclusions

Regarding data compilation, the greatest success of this work has been the application of the ideas of [8] to a variety of problems using `extend-syntax` in Scheme. The solutions we have obtained are concise, are of optimal complexity, and yet are relatively free of data structure considerations including boundedness and sparsity. The potential for solving a wider variety of problems in this style by adding features to Scheme has been shown to be great. We have proposed some new features and modifications to existing features which will be needed to manage data compilation more efficiently.

Before formalization and consideration of semantics can be attempted, the full potential of this methodology must be realized. Thus, most of the time spent on this work has been used to explore applications. An ambitious but important practical application is Logic Programming. Thus, we have attempted to model and implement Prolog with data compilation in mind. Some data compilation is already a part of some of the faster implementations of Prolog although it is not referred to as such. We have begun to explore the potential of data compilation in Prolog and expect to learn much about providing additional tools from it. An application that has been surprisingly resistant to our techniques is testing for graph planarity. The graph planarity test of Tarjan has the characteristics

of depth-first-search and our methods seem most applicable to algorithms which are based on depth-first-search. However, at the moment we cannot offer a good, efficient solution in our methodology. Clearly, we must continue to learn the limits of the methodology by exploring more applications before we can propose what is needed to extend the methodology to be as general as possible.

Other work not directly related to data compilation but supported in part by this grant includes: a workshop in Ulm, Germany, relating Boolean logic, and Logic Programming to Artificial Intelligence; guest editorship of a special issue of *Discrete Applied Mathematics* devoted to connections between logic and combinatorics; a book chapter on the Scheme programming language; and some papers on the probabilistic analysis of algorithms.

9. Publications Under The Grant

Research covered by the grant is reported, at least in part, in the following eight refereed publications.

1. "The Scheme Programming Language," John franco, Daniel P. Friedman, and Olivier Danvy, in *A Comparative Study of Parallel Programming Languages: The Salishan Problems*, John Feo (ed.), to appear in 1991.

2. "Elimination of infrequent variables improves average case performance of Satisfiability algorithms," John Franco, to appear in *SIAM Journal on Computing.*

3. "On the occurrence of null clauses in random instances of satisfiability," John Franco, to appear in *Discrete Applied Mathematics.*

4. "Average case analysis of hashing with lazy deletions," John Franco and Pedro Celis, to appear in *Information Sciences.*

5. "Probabilistic analysis of algorithms for stuck-at test generation in PLAs," John Franco, to appear in *Lecture Notes in Computer Science.*

6. "Multi-way Streams in Scheme," John Franco, Daniel P. Friedman, and S. D. Johnson, in *Computer Languages* **15**, (1990), 109–125.

7. "Towards a facility for lexically scoped, dynamic mutual recursion in Scheme," John Franco and Daniel P. Friedman, in *Computer Languages* **15**, (1990), 55–64.

8. "Creating efficient programs by exchanging data for procedures," John franco and Daniel P. Friedman, *Computer Languages* **14**, (1989), 11–23.

9. "Embedding the Self Language in Scheme," Julia L. Lawall and Daniel P. Friedman in *BIGRE Bulletin*'s special issue on "Putting the Scheme Language to work" (1990), 111–123.

10. *Scheme and the Art of Programming*, George Springer and Daniel P. Friedman, MIT Press and McGraw Hill, (1989).

9. Professional Activities: December 1988 - June 1990

1. Invited visit to the FAW (Research Institute for Applied Knowledge Processing), Ulm, Germany, Summers of 1989 and 1990 (John Franco).

2. Guest Editor: special issue of *Discrete Applied Mathematics* devoted to probabilistic aspects of connections between logic and combinatorics. Targeted for appearance in 1991 (John franco, Mike Dunn, Bill Wheeler).

3. Invited Speaker, *Third IFORS Conference*, Athens, Greece, June, 1990 (John Franco).

4. Invited participant, session chairman *Fifth Advanced Research Institute in Discrete Applied Mathematics*, New Brunswick, New Jersey, May, 1990 (John Franco).

5. Invited Speaker, *14th Symposium on Operations Research*, Ulm, Germany, September, 1989 (John Franco).

6. Discussant at 1989 Meeting of IEEE Standards on Scheme (Daniel P. Friedman).

7. Workshop organizer, *Workshop on Boolean Functions, Propositional Logic and AI Systems*, Ulm, Germany, September, 1989 (John Franco, Franz Josef Radermacher, Michel M. Richter, Peter L. Hammer).

8. Session chair: CORS/TIMS/ORSA meeting of 1989, May 8-10, Vancouver, Canada. Session title is "Probabilistic aspects of Boolean Functions in Operations Research" (John Franco).

9. Invited participant, *Workshop on Mathematical Methods in Artificial Intelligence*, Ulm, Germany, December 1988 (John Franco).

References

[1] Aho, A. V., Hopcroft, J. E., and Ullman, J. D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1974).

[2] Boyer, R. S., and Moore, J. S., "A fast string searching algorithm," *CACM* **20** (1977), pp. 762-772.

[3] Burstall, R. M., and Darlington, J., "A transformation system for developing recursive programs," *J. ACM*, **24**, 1 (January, 1977), pp. 44-67.

[4] Darlington, J., and Burstall, R. M., "A system which automatically improves programs," *Acta Informatica* **6** (1976), pp. 41-60.

[5] Darlington, J., "Program Transformation," in *Functional Programming and its Applications*, Darlington, J., Hendersen, P., and Turner, D., eds., Cambridge University Press (1982).

[6] Dybvig, R. K., *The Scheme Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1987).

[7] Ershov, A. P., "On the essence of compilation," in *Formal Description of Programming Concepts*, E. J. Neuhold, ed., North-Holland (1978), pp. 391-418.

[8] Franco, J., and Friedman, D. P., "Creating efficient programs by exchanging data for procedures," *Computer Languages* **14** (1989), pp. 11–23.

[9] Franco, J., and Friedman, D. P., "Towards a facility for lexically scoped, dynamic mutual recursion in Scheme," *Computer Languages* **15** (1990), pp. 54–64.

[10] Franco, J., and Friedman, D. P., and Johnson, S. D., "Multi-way Streams in Scheme," *Computer Languages* **15** (1990), pp. 109–125.

[11] Franco, J., Friedman, D. P., and Olivier, D., "The Scheme Programming Language," to appear in *A Comparative Study of Parallel Programming Languages: The Salishan Problems*, John Feo (ed.)

[12] Futamura, Y., "Partial computation of programs," *Proc. RIMS Symposia on Software Science and Engineering*, Kyoto, Japan (1982), Springer-Verlag LNCS #147.

[13] Guttag, J. V., Horning, J., and Williams, J., "FP with data abstraction and strong typing," *Proc. 1981 Conference on Functional Programming, Languages, and Computer Architecture*, ACM, New York (1981), pp. 11-24.

[14] Jones, N. D., Sestoft, P., Sondergaard, H., "An experiment in partial evaluation: the generation of a compiler generator," *Proc. 1st Intl. Conf. on Rewriting Techniques and Applications*, Dijon, France (1985), Springer-Verlag, LNCS #202, pp. 124-140.

[15] Knuth, D., *Fundamental Algorithms: The Art of Computer Programming*, Addison-Wesley (1968), pp. 259-265.

[16] Knuth, D., Morris, J. H., and Pratt, V. R., "Fast pattern matching in strings," *SIAM*

J. Comput. **6** (1977), pp. 323-349.

[17] Kohlbecker, E., *Syntactic Extensions in a Lexically Scoped Language*, Ph.D. dissertation, Indiana University, 1986.

[18] Kohlbecker, E., Friedman, D. P., Felleisen, M., and Duba, B., "Hygienic macro expansion," *Proc. of the 1986 ACM Conf. on Lisp and Functional Programming*, (January, 1986), pp. 151-161.

[19] Kohlbecker, E., and Wand, M., "Macro-by-example: deriving syntactic transformations from their specifications," *Conf. Rec. 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, (Munich, January, 1987), pp. 77-84.

[20] Le Metayer, D., "ACE: an automatic complexity evaluator," *ACM Transactions on Programming Languages and Systems* **10** (1988), pp. 248-266.

[21] Lombardi, L. A., "Incremental computation," in *Advances in Computers, Vol. 8*, F. L. Alt, and M. Rubinoff, eds., Academic Press (1967), pp. 247-333.

[22] Scherlis, W. L., "Program improvement by internal specification," *Conf. Rec. 8th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, (Williamsburg, VA, January, 1981), pp. 41-49.

[23] Turchin, V. F., "Program transformation by supercompilation," *Proc. Programs as Data Objects*, Copenhagen, Denmark (1985), Springer-Verlag, LNCS #217, pp. 257-281.